

Lehrstuhl für Informatik 10 (Systemsimulation)



C++ Design Patterns: The Sandwich Pattern

Klaus Iglberger, Ulrich Rude

C++ Design Patterns: The Sandwich Pattern

Klaus Iglberger, Ulrich Rde

Lehrstuhl fr Systemsimulation
Friedrich-Alexander University of Erlangen-Nuremberg
91058 Erlangen, Germany

klaus.iglberger@informatik.uni-erlangen.de

January 27, 2010

In this report we introduce the C++ design pattern called “Sandwich Pattern”. This design pattern is a variation of plain static polymorphism to statically replace certain static and dynamic characteristics of a data type while keeping certain other characteristics unchanged. This design pattern is for example applied in the *pe* rigid multibody physics engine to statically adapt rigid bodies and contacts between these bodies to specific algorithms that are selected at compile time.

1 Motivation

In certain situations during code development the necessity arises to customize a type depending on a certain configuration. It may be necessary to completely replace certain member data and functions of the type, where other parts should remain a permanent component of the type. It may even be necessary to replace a set of virtual functions by a different set of virtual functions to accommodate for a different configuration.

One of these situations was encountered for the implementation of a **RigidBody** class that represents the base for all kinds of rigid, completely undeformable bodies within a framework for the simulation of such rigid bodies. Depending on a set of statically selected algorithms working with the **RigidBody** type, certain member data and functions of the **RigidBody** class should be customizable, whereas other components should not be interchangeable. Imagine, that all of these algorithms pose different requirements on the rigid body data structure. Each algorithm requires different, individual data members within each rigid body in order to be able to optimally perform its task. Additionally, each algorithm requires a certain set of special functions to perform specific tasks or just to access the individual data members. Therefore the question arises how to implement the basic rigid body data structures to accommodate all requirements for the current algorithms and all future algorithms with yet unknown requirements that will have to be integrated into the rigid body framework. Due to the real-time demands of certain simulation scenarios, these data structures additionally have to be suited for high performance computing, i.e., the data structures themselves should not be too costly in terms of performance.

For the remainder of this report, we will use this **RigidBody** class to illustrate the development of a very flexible design pattern called the “Sandwich Pattern”. This C++ design pattern in its basic form allows to statically change certain characteristics (member data and functions) of a data type while keeping certain others unchanged by a clever combination of static and dynamic polymorphism. In Section 2 we will first introduce the general idea of this design pattern by means of the **RigidBody** class. Section 3 will introduce the hierarchical application of this pattern that in particular allows to statically change the dynamic characteristics of the data type, i.e., to customize the set of available virtual functions. To

demonstrate the design pattern Section 4 will show an excerpt of the implementation of the Sandwich Pattern as realized in the *pe* rigid multibody physics engine. Section 5 will conclude the report.

2 The Sandwich Pattern

A first naive approach to solve the problem would be to add all according necessary data and (virtual) function members to the `RigidBody` base class:

Listing 1: Naive approach to integrate the requirements of numerous algorithms into the `RigidBody` class

```
1 class RigidBody
2 {
3     private:
4         // Data members for algorithm 1
5         // Data members for algorithm 2
6         // ...
7
8         // Private (virtual) functions for algorithm 1
9         // Private (virtual) functions for algorithm 2
10        // ...
11
12    public:
13        // Public (virtual) functions for algorithm 1
14        // Public (virtual) functions for algorithm 2
15};
```

This data structure would certainly only result in negligible performance penalties. However, obviously this approach leads to a bloated `RigidBody` base class, both in terms of available functionality as well as in terms of the memory footprint of a `RigidBody` object. All data members of every algorithm are constantly available, although only one algorithm of each category is active at a time. In fact, in our case the configuration consisting of a certain set of algorithms is selected at compile time since switching between the algorithms only very rarely makes sense (for instance because either real-time aspects or physical accuracy aspects are predominant aspects).

One approach to solve customization problems is the use of static polymorphism via the C++ template mechanism:

Listing 2: Template-based implementation of the `RigidBody` class

```
1 template< typename C > // Type of the configuration
2 class RigidBody
3 {
4     private:
5         // Configuration-specific data members
6         // ...
7
8         // Configuration-specific private (virtual) functions
9         // ...
10
11    public:
12        // Configuration-specific public (virtual) functions
13        // ...
14};
```

With this approach it is possible to specifically add the required data and (virtual) function members to the `RigidBody` class depending on the type of the configuration (that represents the set of selected algorithms). Additionally, there is no performance penalty involved in this design. However, the downside of this approach is obvious: Since every specialization of the `RigidBody` class template, which adapts the type to the requirement of a specific configuration/algorithm, additionally needs to define the set of permanent members that should be a constant part of all rigid bodies, the implementation overhead of each specialization might be substantial. Depending on the complexity of the base class, this might very well involve several thousand lines of code that have to be rewritten for every single specialization, i.e., for every new algorithm. However, a simple extension based on dynamic polymorphism improves the situation considerably:

Listing 3: Refactoring of the template-based RigidBody class

```
1 class RigidBodyBase
2 {
3     private:
4         // Common data members for all rigid bodies
5         // ...
6
7         // Common private (virtual) functions for all rigid bodies
8         // ...
9
10    public:
11        // Common public (virtual) functions for all rigid bodies
12        // ...
13 };
14
15 template< typename C > // Type of the configuration
16 class RigidBody : public RigidBodyBase
17 {
18     private:
19         // Configuration-specific data members
20         // ...
21
22         // Configuration-specific private (virtual) functions
23         // ...
24
25     public:
26         // Configuration-specific public (virtual) functions
27         // ...
28 };
```

Due to this refactoring via the introduction of the `RigidBodyBase` class which now contains all common functionality for all rigid bodies, the `RigidBody` class template has only to implement the configuration-specific data and (virtual ¹) function members that are required by the selected algorithms. Additionally, even due to the addition of the `RigidBodyBase` class, no additional performance bottleneck is introduced in the design. Therefore apart from some minor criticisms about this design ² this approach is already well suited for the requirements on the `RigidBody` data structure. However, one additional improvement can still be done. Certain common aspects of the data type to be customized might not fit into the new base class, but should remain in the top level class. Therefore, in addition to the `RigidBodyBase` class we introduce another class that will represent the top level class of this hierarchy and we will additionally slightly adjust the names of the classes:

Listing 4: Application of the Sandwich Pattern to the RigidBody base class

```
1 typedef /* ... */ Config;
2
3 class RigidBodyBase
4 {
5     private:
6         // Common data members for all rigid bodies
7         // ...
8
9         // Common private (virtual) functions for all rigid bodies
10        // ...
11
12    public:
13        // Common public (virtual) functions for all rigid bodies
14        // ...
15 };
16
17 template< typename C > // Type of the configuration
18 class RigidBodyTrait : public RigidBodyBase
19 {
20     private:
21         // Configuration-specific data members
22         // ...
23
24         // Configuration-specific private (virtual) functions
25         // ...
26 }
```

¹We will see the implication and possibilities of the declaration of configuration-specific virtual functions in Section 3.

²See [5] for a discussion about the usage of public virtual functions.

```

27 public:
28     // Configuration-specific public (virtual) functions
29     // ...
30 };
31
32 class RigidBody : public RigidBodyTrait<Config>
33 {
34 private:
35     // Top level common data members for all rigid bodies
36     // ...
37
38     // Top level common private (virtual) functions for all rigid bodies
39     // ...
40
41 public:
42     // Top level common public (virtual) functions for all rigid bodies
43     // ...
44 };

```

In this class hierarchy the Sandwich Pattern is already fully realized. The `RigidBodyBase` class contains all data members and (virtual) functions common to all rigid bodies. On top of that the `RigidBodyTrait` class provides all configuration-specific data and functionality of the rigid bodies. The top level `RigidBody` class contains all common top level data and functionality that (either logically or from an implementation point of view) cannot be moved to the `RigidBodyBase` class. Note that the `RigidBody` class publicly derives from a specific instantiation of the `RigidBodyTrait` class template according to a configuration `Config` that is specified at compile time.

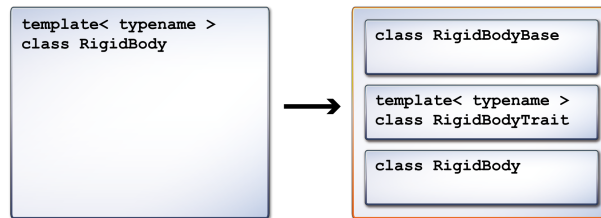


Figure 1: Application of the sandwich pattern to the `RigidBody` base class.

With Figure 1 it becomes obvious that the Sandwich Pattern is a variant of plain static polymorphism. Instead of defining a single class template that would have to carry the burden to redefine common functionality with every specialization, two additional classes are defined, one as the base class of the center customization class, and one as the top level class. All three classes build a logical compound and still represent the original `RigidBody` class, but offer a much more convenient way to adapt the class to specific settings. Especially, specializing for a particular algorithm requires much less programming effort than the single template class.

Although the scheme fulfills all requirements on the `RigidBody` data structure, several potential disadvantages of the Sandwich Pattern should be discussed. One disadvantage in term of performance might be the additional overhead due to two more constructor and destructor calls during the construction and destruction of a type using this design pattern. Whether or not this results in a real performance bottleneck strongly depends on the type of applications the type is involved in. In case such types are frequently created and destroyed and in case the additional function calls are expensive, the performance penalty might be measurable. However, if either construction/destruction is a rare event (as it is for example in rigid body simulations) or in case the constructors and (non-virtual) destructors can be inlined these additional function calls will not be a performance issue. Another disadvantage is the increased depth of the class hierarchy, which is usually considered bad in class design (for a thorough discussion on this topic see [1]). However, in this case the three classes build a logical, inseparable unit, i.e. they represent a single class and are only refactored into three classes for implementation reasons. Therefore, this design arguably does not increase the depth of the class hierarchy.

Another disadvantage from a class design point of view is the potential misuse of the “internal” classes (in this example the `RigidBodyBase` and `RigidBodyTrait` class). It is for instance possible to derive other classes from one of these two internal classes since there is no general protection against inheri-

tance. However, in this case one has to distinguish between “protecting against Murphy, versus protecting against Machiavelli” [2]. However, whereas the intentional misuse cannot be prevented (Machiavelli), the accidental misuse via inheritance is rather improbable (Murphy).

The most important drawback of this design pattern, however, are its implications to the rest of the implementation that uses the customized types. Every functionality that uses an algorithm-specific characteristic of the data type has to be transformed into a template, too. This is in particular true for all algorithms using the `RigidBody` class, since the customized traits classes provide the functionality for these algorithms. Since certain characteristics (as for instance functions) might be completely replaced, using these characteristics even in case a different instance of a trait class is selected will result in compiler errors. Assuming that a class called `Algorithm` expects a `RigidBody` to have a function called `doSomething`, the following example illustrates this problem:

Listing 5: Non-template implementation of Algorithm

```
1 class Algorithm
2 {
3     public:
4         // ...
5         void runAlgorithm( RigidBody* body )
6         {
7             // ...
8             body->doSomething(); // <- This might result in an error even if Algorithm
9                                 //      is not selected
10            // ...
11        }
12        // ...
13};
```

The code in Listing 5 will only compile, if the according traits classes are selected that provide the rigid bodies with the configuration-dependent function `doSomething`. In case a different configuration is selected, the compiler will flag this as an error, since it is possible to determine that a rigid body does not have the required function. Therefore, the use of these custom characteristics has to be hidden before the compiler by use of template argument dependent types. The simplest approach in this example would be to pass the type of rigid bodies as template arguments:

Listing 6: Template-based implementation of Algorithm

```
1 template< typename BodyType > // Type of the rigid bodies
2 class Algorithm
3 {
4     public:
5         // ...
6         void runAlgorithm( BodyType* body )
7         {
8             // ...
9             body->doSomething(); // <- This will always compile since the check for a valid
10                                //      function call is deferred until Algorithm is instantiated
11            // ...
12        }
13        // ...
14};
```

In this case, the compiler has to defer the evaluation of the correctness of the function call until the type of the rigid body is known. In case the `Algorithm` class is not selected, this therefore does not result in an error anymore.

The severity of this disadvantage completely lies in the eye of the beholder. Depending on the situation, it might be considered a minor inconvenience or might even lead to an immediate rejection of the design pattern. Whether or not the Sandwich Pattern can be applied in a certain situation is therefore completely depending on the actual problem.

3 The Hierarchical Sandwich Pattern

It is obviously also possible to apply the Sandwich Pattern on several levels of a class hierarchy. For instance, the design pattern could also be applied to classes deriving from the `RigidBody` class. Let's

assume that the `Sphere` class represents a spherical rigid body and derives publicly from the `RigidBody` class. By applying the Sandwich Pattern, the implementation of the original `Sphere` class would be refactored as following:

Listing 7: Application of the Sandwich Pattern to classes derived from `RigidBody`

```

1 typedef /* customization type */ Config;
2
3 // ... The classes RigidBodyBase, RigidBodyTrait, and RigidBody are implemented as before
4
5 class SphereBase : public RigidBody
6 {
7     private:
8         // Common sphere-specific private data members and functions
9         // Sphere-specific implementation of private virtual functions
10
11     public:
12         // Common sphere-specific public functions
13         // Sphere-specific implementation of public virtual functions
14 };
15
16 template< typename C > // Type of the configuration
17 class SphereTrait : public SphereBase
18 {
19     private:
20         // Configuration-specific private sphere data members and functions
21         // Implementation of configuration-specific private virtual functions
22
23     public:
24         // Configuration-specific sphere public functions
25         // Implementation of configuration-specific public virtual functions
26 };
27
28 class Sphere : public SphereTrait<Config>
29 {
30     private:
31         // Top-level sphere-specific private data members and functions
32
33     public:
34         // Top-level sphere-specific public functions
35 };

```

In case the Sandwich Pattern is applied in this way, it is not only possible to change certain static characteristics of a type (i.e., replace a certain set of member data and functions), but it is also possible to customize the dynamic characteristics of the type by completely replacing a set of virtual functions by another set. Depending on the configuration, each specialization of the `RigidBodyTrait` class can individually define a certain number of virtual functions, that are implemented in the according trait class of the derived class depending on the characteristics of both the derived type and the configuration. Figure 2 gives an impression of the hierarchical application of the Sandwich Pattern to a class hierarchy based on the `RigidBody` base class:

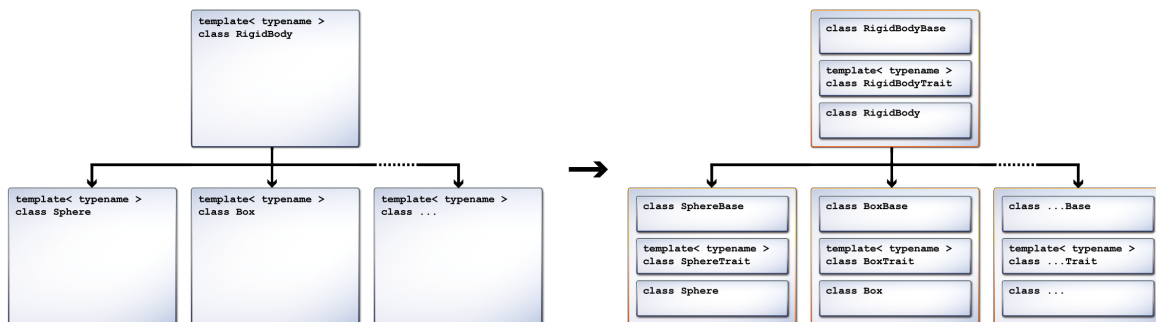


Figure 2: Hierarchical application of the Sandwich Pattern to the class hierarchy based on the `RigidBody` class.

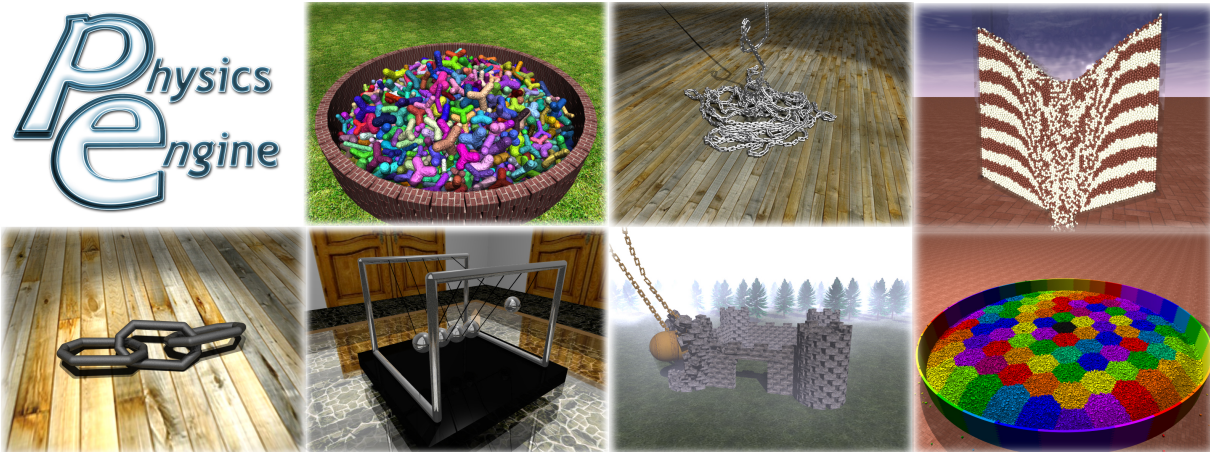


Figure 3: Impressions of *pe* simulations from small-scale scenarios with only several bodies to large-scale simulations involving millions of interacting rigid bodies.

4 Application of the Sandwich Pattern in the *pe* Physics Engine

The *pe* rigid multibody physics engine is a C++ framework for the simulation of rigid, completely undeformable bodies (for more information on *pe* see [4, 3]). One of the major goals of this framework is to provide a large selection of different algorithms for all phases of a time step in the rigid body simulation. This includes different algorithms for

- (a) the (coarse and fine) collision detection phase,
- (b) the contact grouping (batch generation) phase, and
- (c) the collision resolution phase.

The algorithms in each category differ in the way they address the different problems. For instance, for certain simulations it is important to resolve collisions as physically accurate as possible whereas in other simulations the real-time characteristics of the algorithms (i.e. the fast but less accurate processing of collisions) are the crucial requirement. It can be easily seen that therefore the requirements on the `RigidBody` data structure changes depending on which algorithm is selected. The selection of the different algorithms happens in a configuration file:

Listing 8: Compile time selection of rigid body algorithms

```

1 #define pe_COARSE_COLLISION_DETECTOR    pe::detection::coarse::HashGrids
2 #define pe_FINE_COLLISION_DETECTOR      pe::detection::fine::MaxContacts
3 #define pe_BATCH_GENERATOR               pe::batches::UnionFind
4 #define pe_CONTACT_SOLVER                pe::response::FFDSolver

```

For all four categories, an algorithm has to be defined. For instance, the `FFDSolver` is for this example the selected contact solving algorithm. The selection of all algorithms is combined via the `Configuration` class to the compile time configuration `Config`, which is used to instantiate all traits classes:

Listing 9: Definition of the Configuration class

```

1 // CD: Type of the coarse collision detection algorithm
2 // FD: Type of the fine collision detection algorithm
3 // BG: Type of the batch generation algorithm
4 // CR: Type of the collision response algorithm
5 template< template<typename> class CD
6           , typename FD
7           , template<typename> class BG
8           , template<typename,typename,typename> class CR >
9 struct Configuration
10 { /* ... */ };
11
12 typedef Configuration< pe_COARSE_COLLISION_DETECTOR
13                       , pe_FINE_COLLISION_DETECTOR
14                       , pe_BATCH_GENERATOR
15                       , pe_CONTACT_SOLVER
16                       > Config;

```

Listing 10 shows an excerpt of the actual realization of the Sandwich Pattern within the *pe* framework by means of the `RigidBody` base class and the therefrom derived `Sphere` class:

Listing 10: Application to the Sandwich Pattern to the `RigidBody` class hierarchy

```

1 class RigidBodyBase
2 {
3     private:
4         real mass_; // The total mass of the rigid body
5         Vec3 gpos_; // The global position of the center of mass of this rigid body
6         Vec3 v_;    // The linear velocity of this rigid body
7         Vec3 w_;    // Angular velocity of this rigid body
8         // ...
9
10    public:
11        inline real getMass() const { return mass_; }
12        inline const Vec3& getPosition() const { return gpos_; }
13        inline const Vec3& getLinearVel() const { return v_; }
14        inline const Vec3& getAngularVel() const { return w_; }
15        // ...
16 };
17
18 template< typename C >
19 class RigidBodyTrait : public RigidBodyBase
20 {
21     public:
22         // ...
23         virtual void move( real dt ) = 0;
24 };
25
26 class RigidBody : public RigidBodyTrait<Config>
27 {
28     private:
29         RigidBody* sb_; // The superordinate rigid body
30         // ...
31
32     public:
33         // ...
34 };
35
36 class SphereBase : public RigidBody
37 {
38     private:
39         real radius_;
40         // ...
41
42     public:
43         inline real getRadius() const { return radius_; }
44         // ...
45 };
46
47 template< typename C >
48 class SphereTrait : public SphereBase
49 {
50     public:
51         // ...
52         virtual void move( real dt ) { /* Sphere-specific implementation */ }
53         // ...
54 };
55
56 class Sphere : public SphereTrait<Config>
57 {
58     public:
59         // ...
60
61     private:
62         // ...
63         friend Sphere* createSphere( size_t uid, const Vec3& gpos, real radius,
64                                     MaterialID material, bool visible );
65 };

```

In contrast to the abstractions of the previous sections, in this example certain parts of the implementation are shown to demonstrate the use of the Sandwich pattern. The `RigidBodyBase` class contains numerous common data members, as for instance the mass, the global position, the linear and angular

velocity of a rigid body, along with the necessary functionality to access those values. These values are common to all rigid bodies and are independent of the functionality contained in the customizable functionality in the next class of the inheritance hierarchy. The `RigidBodyTrait` class, which inherits all these common characteristics, declares the `move` function that is as default used to move rigid bodies according to their current velocity and the acting forces for a time interval of `dt`. The `RigidBody` class, that publicly derives from the `RigidBodyTrait` instance for the current configuration, only contains common functionality that cannot be moved to the `RigidBodyBase` class. For instance, it contains a handle to a superordinate rigid body of type `RigidBody*`, which refers to a body that may contain this particular rigid body.

`SphereBase` represents the base class for a spherical primitive and as such defines the necessary geometric information (in this case the radius of the sphere) along with the according access function. The `SphereTrait` class template implements the `move` function that was introduced by the according trait class of `RigidBody` for spherical primitives. On the top-level of the hierarchy, the `Sphere` class provides access for the according sphere setup function.

Depending on the selection of the algorithms (see Listing 8), the according specialization of the trait classes is selected. In case of the `FFDSolver`, the `move` function does not suit the requirements and a slightly different behavior is required (for more information see [3]). Therefore a specialization is created, which defines a different set of virtual functions:

Listing 11: Specialization of trait classes for the `FFDSolver`

```

1 // CD: Type of the coarse collision detection algorithm
2 // FD: Type of the fine collision detection algorithm
3 // BG: Type of the batch generation algorithm
4 // C : Type of the configuration
5 template< template<typename> class CD
6           , typename FD
7           , template<typename> class BG
8           , template< template<typename> class
9             , typename
10              , template<typename> class
11              , template<typename,typename,typename> class
12              > class C >
13 class RigidBodyTrait< C<CD,FD,BG,response::FFDSolver> > : public RigidBodyBase
14 {
15     public:
16         // ...
17         virtual void firstPositionHalfStep ( real dt ) = 0;
18         virtual void secondPositionhalfStep( real dt ) = 0;
19         // ...
20 };
21
22 // ...
23
24 // CD: Type of the coarse collision detection algorithm
25 // FD: Type of the fine collision detection algorithm
26 // BG: Type of the batch generation algorithm
27 // C : Type of the configuration
28 template< template<typename> class CD
29           , typename FD
30           , template<typename> class BG
31           , template< template<typename> class
32             , typename
33              , template<typename> class
34              , template<typename,typename,typename> class
35              > class C >
36 class SphereTrait< C<CD,FD,BG,response::FFDSolver> > : public SphereBase
37 {
38     public:
39         // ...
40         virtual void firstPositionHalfStep ( real dt ) { /* Sphere-specific implementation */ }
41         virtual void secondPositionhalfStep( real dt ) { /* Sphere-specific implementation */ }
42         // ...
43 };

```

In the according specializations of the `RigidBodyTrait` and the `SphereTrait` class, the `move` function is completely replaced by the two functions `firstPositionHalfStep` and `secondPositionHalfStep`, which

perfectly suit the requirements of the **FFDSolver**. Please note that only due to the application of the Sandwich pattern we are able to replace a single virtual function with two virtual functions. Therefore the data structures are nicely adapted to the compile time selection of the applied algorithms.

5 Conclusion

In this report, we have introduced the C++ design pattern called “Sandwich Pattern”. Via this pattern it is possible to statically replace a specific set of static and dynamic characteristics of a data type while leaving certain other characteristics untouched. We have demonstrated the application of this design pattern by means of an extended example of the *pe* physics engine, where rigid bodies are statically adapted to a compile time selection of algorithms working on these rigid bodies.

References

- [1] S.C. Dewhurst. *C++ Gotchas*. Addison-Wesley Professional Computing Series. Addison-Wesley, 2007.
- [2] H. Sutter. *Exceptional C++*. C++ In-Depth Series. Addison-Wesley, 2008.
- [3] K. Iglberger and U. R  de. Massively parallel rigid body dynamics simulations. *to be published in Computer Science - Research and Development*, 2009.
- [4] K. Iglberger and U. R  de. The *pe* Rigid Multi-Body Physics Engine. Technical Report 09-9, University of Erlangen-Nuremberg, Computer Science 10 – Systemsimulation, 2009.
- [5] S. Meyers. *Effective C++*. Addison-Wesley Professional Computing Series. Addison-Wesley, third edition, 2008.